VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy rozdělení uzlu v R-stromech

Node Splitting Algorithms in the R-trees

2019

Antonín Kovařík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Antonín Kovařík**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Algoritmy rozdělení uzlu v R-stromech
Node Splitting Algorithms in the R-trees

Jazyk vypracování: čeština

Zásady pro vypracování:

R-strom je vícerozměrná datová struktura, která zůstává od svého prvního představení nejpopulárnější datovou strukturou pro indexování prostorových dat. Tato datová struktura shlukuje vícerozměrná data do regionů, přičemž při naplnění kapacity regionu dochází k jeho rozdělení. Způsob rozdělení regionu (uzlu) má výrazný vliv na kvalitu struktury a následně výkon dotazování nad ní. Cílem této práce je nastudovat, naimplementovat a porovnat algoritmy rozdělení uzlů.

Úkoly:
1. Nastudovat vícerozměrnou datovou strukturu R-strom.
2. Nastudovat algoritmy rozdělení uzlů pro R-stromy.
3. Začlenit algoritmy rozdělení uzlů do stávající implementace.
4. Porovnat algoritmy rozdělení uzlů a jejich vliv na propustnost operací nad R-stromem.

Seznam doporučené odborné literatury:

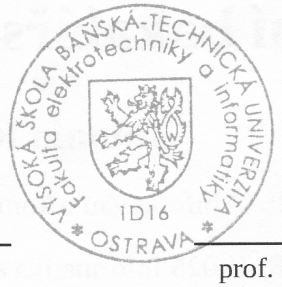
[1] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 47–57. ACM, 1984.
[2] Y. J. Garcıa, M. A. Lopez, and S. T. Leutenegger. On Optimal Node Splitting for R-trees. In Proceedings of the 24th International Conference on Very Large Data Bases (VLDB), pages 334–344. Morgan Kaufmann Publishers Inc., 1998.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Peter Chovanec, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019 .....

Mé poděkování patří Ing. Petrovi Chovancovi Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování bakalářské práce věnoval.

Abstrakt

R-strom je vícerozměrná datová struktura, která je dodnes velmi populární a využívaná struktura pro indexaci prostorových dat. Tato bakalářská práce se zabývá popisem, implementací a nasazením algoritmů pro rozdělování uzlů v R-stromě a jejich dopadem na výkon operací nad touto strukturou. Nachází se zde její podrobný popis a rozebereme její základní operace. Zmíním zde dvě varianty R-stromu, a to R^+-strom a R*-strom. Dále je v práci popsán databázový framework RadegastDB, který obsahuje implementaci R-stromu a u kterého je zkoumán dopad rozdělovacích algoritmů na výkon operací prováděných nad R-stromem a jeho vlastností. Je zde také ukázka implementace vybraných algoritmů v jazyce C++, u kterých následně otestuji jak se liší propustnost operací a vlastnosti struktury při použití vybraných algoritmů.

Klíčová slova: R-strom, RadegastDB, datová struktura, rozdělovací algoritmy

Abstract

R-tree is multi-dimensional data structure, which is still very popular and used structure for indexing spatial data nowadays. This bachelor thesis deals with description, implementation and deployment of node splitting algorithms in R-tree and its impact on performance of operations on the structure. The thesis contains description of the data structure in detail and description of basic operations with this structure. I will also mention description of the two variants of the R-tree, R^+-tree and R*-tree. It also contains description of database framework RadegastDB, which contains implementation of the tree and where is research of algorithms impact on its performance and properties. There is a sample of implementation of some algorithms in C++, in which I will test the difference in throughput of operations and properties with usage of choosen algorithms

Key Words: R-tree, RadegastDB, data structure, splitting algorithms

Obsah

Seznam použitých zkratek a symbolů

MBR	–	Minimum Bounding Rectangle
RAM	–	Random Access Memory
CPU	–	Central Processing Unit

Seznam obrázků

Seznam tabulek

Seznam algoritmů

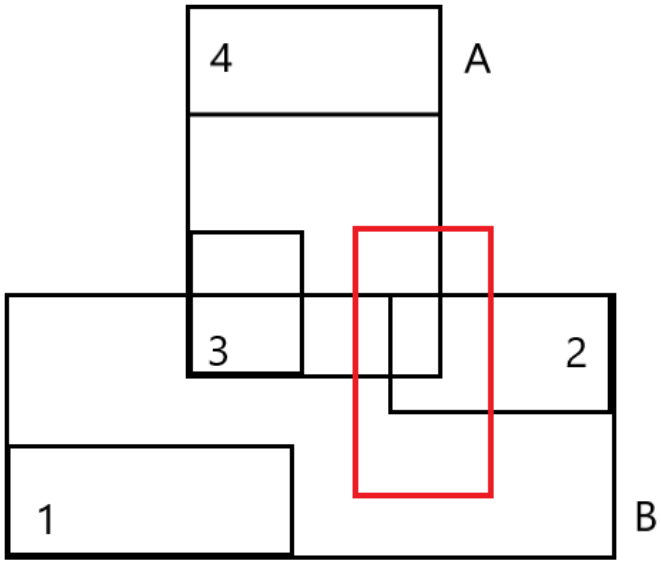
1 Úvod

V dnešním světě je práce s různými druhy dat na denním pořádku. Jedním z typů dat jsou data vícerozměrná, které nejdou vhodně reprezentovat v jednorozměrném prostoru. Tyto prostorové data obsahují informace o objektech v reálném světě. Například to mohou být informace o poloze měst, inženýrských sítí, seznamy památek, plány budov a také mohou zachycovat tvary objektů. Pro lepší práci s těmito daty v roce 1984 představil pan Guttman novou strukturu právě pro indexaci těchto typů dat, R-strom [1]

Do dnes je velmi používaný a od jeho vzniku byly vytvořeny různé varianty R-stromu. Ovšem nejzásadnější vliv na výkonnost této datové struktury má prostorové rozdělení jeho dat do uzlů. Z toho důvodu se po nastudování datové struktury a jejich algoritmů budu zabývat a implementovat rozdělovací algoritmy, které byly navrženy při vzniku R-stromu a jeho následnými variantami a začlením je do frameworku RadegastDB. Poté budu testovat vliv na vlastnosti R-stromu a výkonnost operací vkládání a dotazování se na data uložené v této struktuře a porovnávat rozdíly mezi nimi.

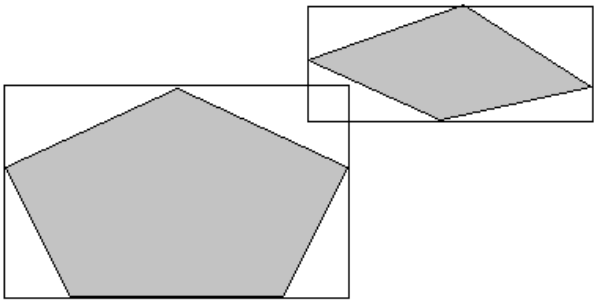
2 R-strom

R-strom [1] je výškově vyvážený strom, který rozšiřuje B-strom o prostrové indexování. Tato datová struktura slouží pro indexaci prostorových dat, které ohraničuje pomocí tzv. minimálních ohraničujících obdélníků (dále jen MBR z anglického slova Minimal Bounding Rectangle). Každý uzel je charakterizován MBR, který je pro každou svou dimenzi charakterizován 2 čísly, tzv. *low*, což je číslo ohraničující MBR minimální hodnotou uzlů nacházejících se v MBR v dané dimenzi a tzv. *high*, který naopak ohraničuje maximální hodnotu uzlů. V R-stromě jsou dva typy uzlů, vnitřní a listový. Vnitřní uzly obsahují záznamy, které definují MBR jejich potomka a ukazatel na tohoto potomka, v listových uzlech jsou uloženy záznamy, které obsahují ukazatel na data a MBR ohraničující tyto data. Struktura je navržena tak, aby při hledání bylo navštíveno co nejmenší množství uzlů. Různé dvě MBR se ovšem také spolu mohou překrývat, čímž se zhoršuje efektivita operací nad R-stromem.



Obrázek 1: Rozmístění uzlů v R-stromě v dvourozměrném prostoru a červeně vyznačený rozsahový dotaz

Přestože jsou záznamy uloženy pouze v jednom uzlu, z geometrického hlediska mohou zasahovat také do jiných MBR. To znamená, že při vyhledávání můžeme navštívit mnoho uzlů, do nichž hledaný MBR zasahuje, než ho najdeme. Na obrázku 1 je červeně vyznačen vyhledávaný obdélník, ve kterém je reálně záznam pouze z B, ale i přesto musíme prohledat i uzel A, protože i uzel A indexuje část plochy uzlu 2. Mohou také nastat případy, kdy při reprezentaci geometrických objektů se jejich MBR protínají, ale přitom se neprotínají geometrické objekty v nich uložené, což můžeme vidět na obrázku 2[6]. .



Obrázek 2: Příklad protínajících se MBR

2.1 Definice

Mějme M jako maximální počet záznamů, který je povolený v jednom uzlu R-stromu. Dále mějme $m \leq \frac{M}{2}$ jako parametr, který určuje minimální počet záznamů, který uzel musí obsahovat. Potom R-strom splňuje následující podmínky:

- Každý vniřtní a listový uzel obsahuje mezi m a M záznamů s výjimkou kořene stromu
- Každý listový uzel obsahuje záznam Z definovaný jako (*MBR*, *oID*), kde *MBR* představuje MBR ohraničující n-rozměrný datový objekt a *oID* jeho identifikátor.
- Každý vnitřní uzel obsahuje záznam Z definovaný jako (p, *MBR*), kde *MBR* je nejmenší možný obdélník, který ohraničuje všechny MBR potomka p.
- Kořen stromu má minimálně 2 potomky, pokud to není list.
- Všechny listy jsou ve stejné hloubce.

Mějme R-strom, ve kterém je N záznamů. Potom výška h stromu může být maximálně:

$$\mathrm{h} = (\log_m N) - 1$$

a maximální počet uzlů při této výšce je:

$$\sum_{k=1}^{h} \frac{N}{m^k} = \frac{N}{m} + \frac{N}{m^2} + ... + 1$$

přičemž tento případ v praxi nenastává. Čím lepší je utilizace, což je obsazenost uzlu, tím pomaleji roste výška stromu a velikost datové struktury. S utilizací a výškou potom také souvisí i propustnost operací vykonávaných nad R-stromem. Hodnota m není pevně stanovena a může být měněna v souvislosti s charakterem dat.

2.2 Operace vkládání

Data se v R-stromu vkládají pouze do listových uzlů, ale vzhledem k možným překryvům můžeme mít více možností k vložení dat. Vkládání začíná v kořeni stromu, odkud rekurzivně pokračuje až k listu. Na každé úrovni se projdou jednotlivé MBR a určí se právě jeden, do kterého záznam

vložíme. Rozhoduje se na základě požadavku na co nejmenší rozšíření MBR, abychom předešli velkému prázdnému místu, tzv. *deadspace*, což je prostor v MBR, ve kterém nejsou žádné data a přitom do tohoto MBR náleží. Pokud u dvou či více uzlů bude stejná plocha rozšíření, rozhoduje se na základě počtů záznamu v uzlu a vybere se ten, který má méně záznamů. Jakmile dosáhneme listu, tak musíme ověřit, zda je v listu stále místo pro další záznam, nebo jestli se bude muset provést rozdělení uzlu na dva uzly a jeho záznamy přerozdělit do těchto uzlů. Pokud dojde k rozdělení, nový uzel přidáme do rodičovského uzlu. Rozdělení musíme propagovat až ke kořeni, protože rodičovský uzel může přetéct. Při přetečení kořene se kořen rozdělí a výška stromu se zvýší. Algoritmy pro rozdělení uzlů budou popsány v kapitole 3.

Algoritmus vkládání: Vstup je vkládaný záznam Z a kořen stromu R.

1. Zavoláme metodu *ChooseLeaf(Z, R)*, která nám na základě vkládaného záznamu Z vrátí list L, do kterého vložíme záznam Z
2. Jestliže v L je dostatek místa pro Z, tak do něj záznam vložíme, v opačném případě rozdělíme uzel L na dva *oldL* a *newL* a přerozdělíme do nich záznamy pomocí rozdělovacích metod popsaných v kapitole 3.
3. Propagujeme změny do vyšších úrovní R-stromu metodou *AdjustTree(oldL, null)*, pokud list nebyl rozdělen, v opačném případě voláme metodu s parametry *AdjustTree(oldL, newL)*.
4. Jestliže je potřeba rozdělit i kořen, vytvoříme nový kořen a nastavíme mu jeho potomky.

Metoda ChooseLeaf: Vstup je záznam Z a uzel N. Výstup list L.

1. Nastavíme uzel N jako kořen podstromu.
2. Jestliže je N list, metoda vrátí N.
3. Jesliže N není list stromu, mějme potomka uzlu N, uzel M, jehož MBR se rozšíří nejméně při vložení záznamu Z. Jestliže máme více uzlů se stejným rozšířením plochy, tak vybereme ten uzel M, který má méně záznamů. Zavoláme metodu *ChooseLeaf(Z, M)*

Metoda AdjustTree: Vstupem je uzel *oldN* a *newN*

1. Jestliže je *oldN* kořen, máme hotovo.
2. Mějme rodiče P uzlu *oldN*, potom upravme MBR uzlu *oldN* v rodiči P tak, aby zabíral všechny záznamy uzlu P co nejtěsněji.
3. Jestliže máme i uzel *newN*, který vznikl při rozdělení uzlu na předchozí vrstvě, vytvoříme ukazatel *newNp*, který bude ukazovat na *newN* a *newNMBR*, což bude obdélník, který co nejtěsněji zabírá všechny MBR uzlu *newN*. Poté přidáme tento uzel do P. Pokud není v P místo, použijeme pro rozdělení uzlů z kapitoly 3 a uzel P rozdělíme na uzly P a *newP*.

4. Voláme metodu *AdjustTree(P, newP)* pokud byl rodičovský uzel rozdělen, jinak voláme metodu *AdjustTree(P, null)*.

2.3 Operace smazání

Při mazání záznamu z R-stromu rekurzivně prohledáváme strom od kořene k listu, dokud záznam nenajdeme, a potom ho odstraníme. Při odstranění může nastat podtečení uzlu. Při podtečení uzlu si uložíme všechny jeho záznamy do dočasného listu, ve kterém budeme uschovávat tyto záznamy. Pokud přetečení nenastane, tak upravíme obdélník tak, aby opět co nejtěsněji zabíral všechny MBR v něm. Vracíme se zpět ke kořeni stejnou cestou, a pro každý uzel na této cestě zkontrolujeme, zda nemá méně než m záznamů. Pokud ano, tak si záznamy z tohoto uzlu uložíme do dočasného listu a tento uzel odstraníme z jeho rodičovského uzlu. Takto pokračujeme až ke kořeni. Všechny záznamy z dočasného listu znovu vložíme. Varianta znovuvložení na místo spojení uzlu s druhým uzlem je zvolena kvůli lepšímu uspořádání záznamů a tedy i lepšímu výkonu.

Algoritmus mazání: Vstup je záznam Z a kořen stromu R

1. Použijeme metodu *FindLeaf(Z, R)* na nalezení listu L se záznamem Z.
2. Odstraníme záznam Z z listu L.
3. Použijeme metodu *CondenseTree(L)*.
4. Pokud bude mít kořen pouze jednoho potomka, tak tohoto potomka uděláme novým kořenem.

Metoda FindLeaf: Vstupem je uzel N a záznam Z. Výstupem je list, ve kterém je záznam Z.

1. Jestliže N není list, tak pro každé MBR jeho potomka P vyzkoušej, jestli se překrývá se záznamem Z. Pro všechny tyto záznamy zavoláme metodu *FindLeaf(P, Z)* pro strom, jehož kořen bude záznam právě tohoto potomka P, dokud nenajdeme záznam Z.
2. Jesliže je N list, tak zkontrolujte všechny záznamy, jestli se některý shoduje se záznamem Z. Pokud ano, vrátíme N.

Metoda CondenseTree: Vstupem je uzel N

1. Mějme Q jako množinu listů, jejichž záznamy mají být znovuvloženy.
2. Jestliže N je kořen, přejdeme na poslední krok. Jinak mějme uzel P, který je rodičem N.
3. Pokud N má méně než m záznamů, odstraňme N z P a přidejme ho do množiny Q.

4. Jestli N má dostatek záznamů, upravme jeho MBR, aby co nejtěsněji ohraničovala záznamy v uzlu.

5. Voláme metodu *CondenseTree(P)* .

6. Znovu vložme všechny záznamy z množiny Q, přičemž listové záznamy jsou vkládány do listů a záznamy z vyšších vrstev musejí být vloženy do vrstvy, z které byli odebrány.

2.4 Vyhledávání v R-Stromu

Existují různé typy dotazování nad R-stromem, a to bodové, rozsahové a podobnostní dotazy. Jelikož je nejběžnějším z nich dotazování rozsahové, tak popíšu tento typ. U vyhledávání je vstupem vyhledávaný obdélník. Vyhledávání začíná u kořene stromu. U každého MBR potomka v uzlu se rozhodne, zda se vyhledáváný obdélník překrývá s tímto obdélníkem nebo nikoliv. Jestliže se protínají, tak se bude prohledávat i uzel potomka. Vyhledávání takto pokračuje rekurzivně dál, dokud se neprohledají všechny uzly, se kterými se hledaný obdélník protíná. Jakmile se dosáhne listových uzlů, prohledávají se MBR a pokud MBR leží v rámci vyhledávané oblasti, tak jsou jejich data přidána do množiny vysledků.

Algoritmus vyhledávání: Vstup je kořen R a dotazovaný obdélník QR. Výstupem je množina nalezených záznamů.

1. Jestliže R není list, tak pro každého jeho potomka rozhodneme, zda se jeho MBR protíná s QR. Pro všechny protínající se potomky zavoláme tuto metodu na strom, jehož kořenem bude právě tento potomek.

2. Jestliže je R list, pro všechny jeho záznamy určíme, zda se protínají nebo ne. Jestli ano, tak přidáme tento záznam do množiny výsledků.

2.5 Varianty R-stromu

Z R-stromu vznikly různé varianty, které se snaží zefektivnit tuto strukturu. Princip operací v různých variantách R-stromu se příliš neliší, akorát jsou použita jiná kritéria, podle kterých se rozhoduje o rozdělení záznamů v uzlech. Pouze v operaci rozdělování uzlu se algoritmy liší, některé algoritmy jsou úplně odlišné od původně uvedených panem Guttmanem. Variant R-stromu je spousta, zde popíšu okrajově jen dvě z nich, a to R^+-strom [2] a R*-strom [3].

2.5.1 R$^+$-strom

Při snaze o zlepšení efektivity vyhledávání vznikla varianta R-stromu, tzv. R^+-strom [2], která byla navržena k vyhnutí se prohledávání více větví stromu pri dotazování. Tohoto bylo dosáhnuto tak, že R^+-strom nedovoluje překrývaní uzlů na stejné úrovni. Ovšem toto má za následek, že vkládáné záznamy mohou být rozděleny do dvou nebo více MBR, tedy jsou duplikovány a

uloženy ve více listech. Díky duplikaci bude při vyhledávání záznamu prohledána pouze jedna cesta, což velmi zefektivní vyhledávání. Takže oproti původnímu R-stromu, se zde liší operace vkládání, kde je potřeba záznam vložit do všech uzlů, se kterými se překrývá. Jelikož se mohou překrývat více než dvě MBR, tak se tyto záznamy musí duplikovat do všech uzlů a tento počet narůstá s dimenzí. Tím roste také velikost datové struktury, tedy použitelnost je pouze při nižších dimenzí. U operace odstranění musíme smazat všechny kopie mazaného záznamu, což může mít za následek zhoršení utilizace stromu. Kvůli tomu bude muset probíhat reorganizace stromu. Liší se zde i algoritmus pro rozdělování uzlu, který bude popsán v další kapitole.

2.5.2 R*-strom

R*-strom [3] byl uveden roku 1990 jako další varianta snažící se o zlepšení výkonu původního R-stromu. Dodnes používaný R*-strom se snaží pro zlepšení výkonu brát v potaz více kritérií než původní R-strom a to:

1. Minimalizovat plochu pokrytou každým MBR. Toto kritérium je stejné jako v původním R-stromě a snaží se o minimalizaci tzv. *deadspace*, což je oblast, která je pokryta MBR, ale nejsou zde žádná data.

2. Minimalizace překryvů mezi MBR.

3. Obvod MBR, což vede k tvoření čtvercových MBR, tedy i k menším rozměrům MBR.

4. Utilizace uzlu, což vede k menšímu počtu uzlů, tedy menší struktuře a menší výšky stromu.

Ovšem minimalizovat všechna kritéria zároveň není možné, např. pro minimalizaci plochy je nutné mít volnost ve výběru tvaru MBR, čímž se porušuje třetí kritérium. Tato varianta se oproti původnímu R-stromu liší ve způsobu vkládání záznamu a řešení přetečení uzlu. Při vkládání se ve vnitřních uzlech stejně jako v R-stromě volí uzel, jehož plocha se nejméně rozšíří, ovšem v listových uzlech se snažíme minimalizovat plochu, která se překrývá s ostatními uzly. Další změna je, že při přetečení uzlu se nepoužívá hned rozdělovací algoritmus, ale zkontroluje se, zda některé záznamy v uzlu mohou být přerozděleny do druhého uzlu. Toto se provede znovuvložením části záznamů, u této možnosti je uváděno 30 procent jako dobré řešení. Dále se zde používá i jiný algoritmus pro rozdělení uzlu, který bude zmíněn v další kapitole.

3 Algoritmy pro rozdělení uzlu

Při přetečení uzlu je nutné jej rozdělit. To, jak jej rozdělíme, má velký vliv na vlastnosti a propustnost operací v R-stromě, která se může mnohonásobně lišit. Proto byly při uvedení R-stromu uvedeny spolu s ním 3 různé varianty, jak uzel rozdělit. Pan Guttman prezentoval exponenciální, kvadratický a lineární algoritmus, příčemž se soustředil hlavně na minimalizaci plochy, kterou po rozdělení uzlu na dva uzly budou uzly tvořit. Algoritmy pana Guttmana[1] braly v potaz především minimalizaci plochy, neřešily ovšem disjunktnost uzlů, což je pro rychlost také zásádní. Exponenciální algoritmus, jak již v názvu napovídá, má exponenciální složitost a je nepoužitelný při vyšším počtu záznamů v uzlu, i přes to, že má lepší rozložení záznamů než kvadratický nebo lineární algoritmus. Naproti tomu lineární algoritmus je co se týče vkládání nejrychlejší, ale rozdělení záznamů nebude tak kvalitní. Jako kompromis mezi rychlostí vkládání a rozdělením záznamů je algoritmus kvadratický, který nebude dosahovat takových rychlostí při vkládání jako lineární, ale naproti tomu bude mít lepší rozdělení uzlů a tím se navýší i rychlost dotazování. Dále zde popíšu Greenův algoritmus[3], který byl uveden také mezi prvními. Tyto 4 algoritmy budou dále implementovany a testovány. Mimo to popíšu v této sekci ještě algoritmus pro rozdělování uzlu v R^+-stromě[2] a R*-stromě[3], Hilbertův rozdělovací algoritmus[7, 8] a lineární algoritmus Ang et Tan[6, 9, 10].

3.1 Kvadratický algoritmus

1. Vytvoříme všechny dvojice záznamů, a jako první záznam pro každý z uzlů vybereme tu dvojici, která by vyplítvala nejvíce místa, kdyby byla dána do jednoho uzlu (to vypočítáme odečtením od celkové plochy plochu, kterou zabírá každý záznam).

2. Projdeme zbývající záznamy. Pro každý záznam vždy vypočítáme, o jakou plochu se změní každý z uzlů, kdybychom jej do uzlu přidali. Vybereme ten záznam, který bude mít největší rozdíl mezi plochami, o které by se rozšířil při vložení do uzlů a vložíme jej do uzlu, který se rozšíří méně.

3. V dalším kroku budeme kontrolovat, jestli jeden z uzlu není již tak velký, abychom zbývající počet záznamů museli přiřadit do druhého uzlu, aby splňoval pravidlo minimálního počtu záznamů v uzlu.

4. Toto opakujeme od 2. kroku, dokud nerozdělíme všechny záznamy.

3.2 Lineární algoritmus

1. Nejdříve vybereme první záznamy pro oba uzly. Pro každou dimenzi vybereme obdélník s nejpravějším levým okrajem a nejlevějším pravým okrajem. Zapamatujeme si tyto páry a jejich separaci, tj. rozdíl mezi nejpravějším levým okrajem a nejlevějším pravým okrajem.

Separaci normalizujeme a záznamy s největší normalizací vložíme každý do jednoho uzlu a vytvoříme jejich MBR.

2. Dále vybereme jakýkoliv další záznam, který přiřadíme do toho uzlu, ve kterém plocha naroste méně.

3. V dalším kroku budeme kontrolovat, jestli jeden z uzlů není již tak velký, abychom zbývající počet záznamů museli přiřadit do druhého uzlu, aby splňoval pravidlo minimálního počtu záznamů v uzlu.

4. Opakujeme krok 2 a 3.

3.3 Greenův rozdělovací algoritmus

Greenův rozdělovací algoritmus[3] má taktéž kvadratickou časovou složitost, používá třídící algoritmus pro uspořádání záznamů a poté přerozdělí první polovinu záznamů do jednoho uzlu a druhou polovinu záznamů do druhého uzlu.

1. Vybereme první záznamy do obou uzlů Z_1 a Z_2 tak, jak bylo popsáno v kvadratickém algoritmu.

2. Pro každou dimenzi spočítáme normalizaci těchto záznamů tak, že vydělíme vzdálenost mezi uzly Z_1 a Z_2 délkou pokrývajícího obdélníku v dané dimenzi.

3. Vybereme dimenzi, která má největší normalizaci.

4. Seřadíme záznamy v dané dimenzi podle jejich hodnoty *low*.

5. Přerozdělíme prvních $\frac{(M+1)}{2}$ záznamů do prvního uzlu a zbývajících $\frac{(M+1)}{2}$ záznamů do druhého uzlu

6. Jestliže je M+1 liché číslo, tak zbývající záznam vložíme do toho uzlu, jehož velikost MBR se rozšíří méně.

3.4 Rozdělovací algoritmus v R*-stromě

Rozdělovací algoritmus v R*-stromě[3] je výkonnější než zde dosud popisované algoritmy a je dodnes stále ještě nejpoužívanější varianta pro rozdělování uzlů. V tomto algoritmu seřadíme záznamy v každé dimenzi a následně je zkoušíme přerozdělit do uzlů jak je uvedeno níže. Vybereme tu dimenzi, v které bude obvod MBR nejmenší. Poté přerozdělíme záznamy podle této dimenze tak, abychom dosáhli co nejmenšího překrytí uzlů.

1. Pro každou dimenzi seřadíme záznamy podle hodnoty *low*, potom podle hodnoty *high*, a potom budeme záznamy rozdělovat následovně: Mějme M-2m+2 různých rozdělení definovaných jako (m-1+k), kde k $\in$ (1,.., M-2m+2). První skupina obsahuje prvních (m-1+k) záznamů a druhá skupina zbytek.

2. Spočítáme obvod O těchto MBR. Vyberme tu osu, která má minimální O.

3. Ve vybrané dimenzi zkoušíme rozdělení splňující minimální počet záznamů a rozdělíme záznamy tak, abychom minimalizovali překrytí mezi nimi. Pokud dojde u dvou nebo více rozdělení ke stejné minimalizaci překrytí, tak vybereme to rozdělení, jehož celková plocha bude nejmenší.

4. Rozdělíme první část záznamů do prvního uzlu a druhou část záznamů do druhého uzlu.

3.5 Rozdělovací algoritmus v R$^+$-stromě

Tento algoritmus[2] oproti ostatním verzím R-stromu používá duplikování záznamů, což má za následek, že při dotazování nalezneme hledaný záznam hned v první větvi, kterou prozkoumáváme. Ovšem nevýhodou je, že nám roste velikost stromu a složitost ostatních operací v tomto stromě. Dalším následkem je, že pokud rozdělíme vnitřní uzel, tak musíme opět propagovat rozdělení do těch potomků, kteří zasahují do tohoto rozdělení.

1. Mějme O_x a O_y nejmenší x-ovou resp. y-ovou souřadnice. Podle každé dimenze budeme simulovat rozdělení uzlu, poté vybereme to nejlepší.

2. Seřadíme si záznamy podle dimenze, podle které budeme rozdělení provádět.

3. Začneme od O_x, resp. O_y a bereme další záznamy podle seřazeného listu, dokud nenaplníme uzel m záznamy.

4. Nyní spočítáme cenu tohoto rozdělení podle kritéria, podle kterého rozdělujeme, např. minimální pokrytá plocha uzlu, a dále máme bod, kde došlo k naplnění uzlu a kde se uzel půlí.

5. Vybereme rozdělení, které má nejmenší cenu, a mějme dva regiony S_1 a S_2 jako výsledek tohoto rozdělení.

6. Vytvoříme dva nové uzly n_1 jehož MBR obsahuje region S_1, a n_2, jehož MBR obsahuje region S_2, vzniklé po rozdělení uzlu N.

7. Do nově vytvořených uzlů n_1 a n_2 vložíme všechny záznamy, které leží kompletně v jejich MBR. Pro ty záznamy, které leží na hranici, tedy v obou MBR, provedeme následující:

8. Jestliže je uzel N listový uzel, poté vložíme záznam do obou nových uzlů.

9. Jinak rekurzivně rozdělíme všechny potomky, kteří zasahují do tohoto rozdělení. Nechť máme MBR_1 a MBR_2 dva uzly vzniknuté ze záznamu ležícího na hranici, kde MBR_1, resp. MBR_2 leží celý v uzlu n_1 resp. n_2. Tyto dva vzniklé uzly vložíme do korespondujících uzlů n_1 a n_2

10. Jestliže je N kořen, tak vytvořme nový kořen s potomky n_1 a n_2. Jinak nechť je P rodič uzlu N, potom nahraďme uzel N uzly n_1 a n_2. Jestliže P přeteče, uzel rozdělíme.

3.6 Ang and Tan Lineární algoritmus

Uvažujme dvourozměrný prostor, pro vícerozměrné prostory jsou pro každou další osu použity další dvojice listů a rozdělení probíhá přímočaře, a mějme uzel N, který chceme rozdělit, a dále MBR uzlu N definované jako MBR_N = (Left, Botton, Right, Top). Dále mějme záznamy Z jako potomky uzlu Z definované jako Z = (xl, xh, yl, yh). Dále algoritmus používá 4 listy L_L, L_B, L_R, L_T.

1. Pro všechny záznamy Z uzlu N provedeme následující: Jestliže xl - Left < Right - xh, tak přidáme Z do L_L, jinak Z přidáme do L_R, a jestliže yl - Bottom < yh - Top, tak přidáme Z do L_B, jinak přidáme Z do L_T.

2. Jestliže $\max(|L_L|, |L_R|) < \max(|L_B|, |L_T|)$ rozdělme uzel podél dimenze x, nebo jestliže $\max(|L_L|, |L_R|) > \max(|L_B|, |L_T|)$ rozdělme uzel podle dimenze y. Pokud dojde k rovnosti, tak následně otestujeme překrytí$(|L_L|, |L_R|)$ < překrytí$(|L_B|, |L_T|)$, což povede k rozdělení podle x, nebo překrytí$(|L_L|, |L_R|)$ > překrytí$(|L_B|, |L_T|)$, což povede k rozdělení podle y, a pokud i teď dojde k rovnosti, tak vybereme podle nejmenší plochy pokrývající uzly.

3.7 Hilbertův rozdělovací algoritmus

V Hilbertově stromě se před rozdělením uzlu nejdříve snažíme přerozdělit záznamy přetíkajícího uzlu do vedlejších uzlů, pokud toto není možné, tak použijeme rozdělovací algoritmus:

1. Seřadíme všechny záznamy v uzlu podle Hilbertova čísla.

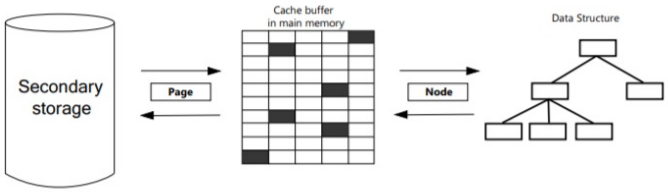
2. Vložíme první polovinu záznamů do původního uzlu a druhou polovinu záznamů vložíme do nově vytvořeného uzlu.

4 Implementace

Po nastudování algoritmů pro rozdělování uzlů bylo dalším cílem této bakalářské práce některé z těchto algoritmů začlenit do stávající implementace frameworku RadegastDB a otestovat propustnost operací při použití implementovaných algoritmů v jazyce C++. Do databázového frameworku byly doimplementovány základní 3 algoritmy od pana Guttmana, tedy exponenciální, kvadratický a lineární. Dále jsem začlenil do stávající implementace Greeneův rozdělovací algoritmus.

4.1 Framework RadegastDB

RadegastDB je databázový framework vyvíjený databázovou skupinou Katedry informatiky na Vysoké škole Báňské - Technické Univerzity Ostrava. Je po spousty let obohacován dalšími algoritmy, datovými strukturami a indexovacími metodami. Tento framework je psán v jazyce C++ a obsahuje širokou škálu indexovacích technik a datových struktur. Součástí RadegastDB je také zmiňovaný R-strom, do kterého byly doimplementovány rozdělovací algoritmy.



Obrázek 3: Struktura RadegastDB

Datové struktury v RadegastDB jsou stránkované. Tedy datová struktura je uložena na úložišti a jen část struktury je uložena v paměti RAM. Na obrázku 3 [4] vidíme, že stránky v RadegastDB reprezentující uzly načítáme do cache. Při požadavku datové struktury o data zkontrolujeme, zda jsou již načtena v cache, pokud ano, tak ji použijeme, v opačném případě načteme nejdříve stránku do cache a potom s ní pracujeme. Stránkování struktur v RadegastDB zastřešuje třída *cQuickDB*, která obsahuje instance třídy *cNodeCache* a *cMemoryPool*. Obsahuje metodu *Create()*, která slouží pro vytvoření persistentního souboru na sekundárním úložišti, vytvoří *NodeCache*, a připraví *QuickDB* k používání. Dále metodu *Open()* pro otevření tohoto souboru a metodu *Close()* ke smazání souboru. Třída *cMemoryPool* slouží ke správě paměti a poskytování paměti pro dočasné proměnné. Třída *cNodeCache* slouží jako cache pro persistentní datovou strukturu umožňující zrychlení přístupu k datům. Tato třída používá také *mutex* pro zamykání dat při přístupu k nim a jejich modifikaci.

4.1.1 Implementace R-stromu

Práce s R-stromem zahrnuje nejen využívání metod, ale také konfiguraci cache a R-stromu. Část možností pro konfigurace vidíme v tabulce 1.

Konstanta	Popis
CACHE_SIZE	Velikost cache
BLOCK_SIZE	Velikost bloku
DSMODE	Typ datové struktury
DIM	Dimenze dat
DATA_LENGTHS	Délka vkládaných dat
SIGNATURE	Určuje, jestli R-strom používá signaturu
ORDERED_RTREE	Určuje,zda je strom uspořádaný
COLLECTION	Použitá kolekce

Tabulka 1: Konstanty pro nastavení R-stromu

V proměnné *Collection* nastavujeme vstupní kolekci dat, která může reprezentovat buď připravená data na disku, nebo můžeme data náhodně vygenerovat pomocí třídy *cTuplesGenerator*. Pro využití této třídy nastavujeme velikost dimenze a počet náhodně vygenerovaných záznamů. Konfiguraci R-stromu uchováváme v instanci třídy *cRTreeHeader*. Ve třídě *cRTreeConst* nastavujeme konstanty pro R-strom a volíme zde rozdělovací algoritmus pro uzly stromu. Pro spuštění R-stromu používáme metodu *SimpleTest()*, která nastaví všechny potřebné parametry a zavolá metodu *Build()* pro sestavení R-stromu a jako parametr přijímá *cRTreeHeader*, ve které jsou všechny informace pro sestavení datové struktury. Pro vkládání záznamu do R-stromu je implementována metoda *Insert()* ve třídě *cCommonRTree_Insert* a pro vyhledávání je implementována metoda *Find()* ve třídě *cCommonRTree_Query*. Algoritmus 1 ilustruje vkládání záznamu do R-stromu. Nejdříve rozhodujeme, do kterého podstromu záznam vložíme (1. řádek), jakmile dosáhneme listového uzlu, tak v případě, že není plný, tak do něj vložíme záznam (15. řádek), v opačném případě jej rozdělíme (17. řádek). Při přetečení listového uzlu vkládáme druhý uzel vzniklý po rozdělení do rodičovského uzlu (12. řádek), ale může nastat také přetečení rodičovského uzlu a k jeho následnému rozdělení (10. řádek). Algoritmus 2 popisuje dotazování v R-stromu, načteme uzel, pokud je vnitřní (2. řádek) tak procházíme všechny jeho potomky (5. řádek), dokud nějaký potomek nezasahuje do *QueryRect* (6. řádek). Na tohoto potomka zavoláme metodu *RangeQuery()* (10. řádek). Jestliže je načtený uzel list a pokud se nějaký jeho záznam protíná s *QueryRect* (13. řádek), tak ho přidáme do výsledku (14. řádek).

Pro měření propustnosti těchto operací nám slouží třída *cTimer*. Pro odstranění záznamu slouží třída *cCommonRTree_Delete*, která implementuje pouze metodu *delete()*. Třída *InsertBuffers* nám slouží k uložení potomků a nalezneme zde také MBR uzlu. Pro práci s MBR je k dispozici třída *cMBRectangle*, která obsahuje metody *GetLoTuple()*, *GetHiTuple()* pro získání *low* a *high* a také metody *SetLoTuple()* a *SetHiTuple* pro nastavení *low* a *high*. Dále jsou k

Algoritmus 1: INSERT

```
   Input: tuple, node1, node2, nodeIndex
 1 n ← ReadNode(nodeIndex);
 2 if N.IsInnerNode() then
 3     order ← N.FindMbr(tuple);
 4     split ← Insert(tuple, N.GetChildIndex(order));
 5     if ¬split then
 6         N.GetMbr(order).ModifyMbr(node1.MBR);
 7     else
 8         N.GetMbr(order).ModifyMbr(node1, MBR);
 9         if N.IsFull() then
10             ret ← N.Split(node2.MBR, node2.index, node1, node2);
11         else
12             N.Insert(node2.MBR, node2.index);
13     else
14         if ¬N.IsFull() then
15             N.Insert(tuple);
16         else
17             ret ← N.Split(tuple, node1, node2);
18     return ret;
```

Algoritmus 2: RANGEQUERY

```
   Input: queryRect, nodeIndex
   Output: resultSet
 1 N ← ReadNode(nodeIndex);
 2 if N.IsInnerNode() then
 3     order ← 0;
 4     while order < N.mItemCount do
 5         for i ← order + 1 to mItemCount do
 6             if IsIntersected(GetItem(i), queryRect) then
 7                 order ← i;
 8                 break;
 9         if order ≤ N.ItemCount then
10             RangeQuery(N.GetChildIndex(order, queryRect);
11 else
12     for i ← order + 1 to mItemCount do
13         if IsIntersected(GetItem(i), queryRect) then
14             resultSet.Add(GetItem(i));
15 return resultSet;
```

dispozici metody *Resize()* pro změnu velikosti, *Copy()* pro kopírování bloku paměti do druhého bloku paměti, metoda *ModifyMbr()*, která bere jako parametr vkládaný záznam a upraví MBR, a také metodu *ModifyMbrByMbr()*, která slouží také k úpravě MBR. Dále jsou zde metody *IsIntersected()*, což ověří, zda se dva MBR protínají, metoda *IsInRectangle()*, což ověří, zda záznam leží v MBR.

Pro reprezentaci záznamů slouží třída *cTuple*. V této třídě nalezneme metody jako *SetValue()*, která bere jako argumenty pořadí, hodnotu a má spoustu přetížení pro více datových typů, jako např. float, double etc., dále pro všechny tyto typy máme i metodu *GetValue()*, která vrátí hodnotu v dané dimenzi. Dále je tu metoda *GetTuple()*, která vrátí celý uzel. Tato třída má také metody pro porovnání, kopírovaní, vytištení uzlu do konzole a veškerou práci s uzly. Pro operace s listovými uzly se používá třída *cRTreeLeafNode*, která obsahuje veškeré metody pro rozdělení uzlu, vytvoření dvou uzlů a vytvoření MBR. Pro práci s vnitřními uzly slouží třída *cRTreeNode*, která obsahuje také algoritmy pro vytvoření a rozdělení MBR.

4.2 Vlastní implementace

Rozdělovací algoritmy byly implementovány do tříd *cRTreeLeafNode* a *cRTreeNode*. Ve třídě *cRTreeLeafNode* se pracovalo se záznamy reprezentovanými třídou *cTuple* a ve třídě *cRTreeNode* se pracovalo se záznamy reprezentovanými třídou *cMBRectangle*. Pro vnitřní i listové uzly byly implementovány algoritmy pro rozdělení uzlu, tedy kvadratický, lineární, exponenciální a Greeneův rozdělovací algoritmus.

4.2.1 Kvadratický algoritmus

Jak již bylo zmíněno v sekci rozdělovacích algoritmů, tento algoritmus má 2 základní kroky při přerozdělování záznamů, a to vybrání základu pro oba uzly, a potom rozdělení záznamů do těchto uzlů. Kvadratický algoritmus bere jako parametry *mbr1Lo*, *mbr1Hi*, *mbr2Lo*, *mbr2Hi* typu char*, do kterých ukládáme MBR prvního a druhého uzlu, dále parametr *buffers* typu *cNodeBuffers*, kde máme uloženy záznamy uzlu a *NodesSelected* typu char*, kde ukládáme rozřazení uzlů. První část algoritmu je ilustrována v algoritmu 3.

Algoritmus 3: PickSeedsQuadratic

```
1 for i ← 0 to mItemCount do
2     for j ← i + 1 to mItemCount do
3         tmp ← ComputeLargestMBR(GetItem(i), GetItem(j));
4         if largestMbr < tmp then
5             largestMbr ← tmp;
6             first ← i;
7             second ← j;
```

Algoritmus 3 popisuje výběr prvních záznamů do uzlu. Algoritmus prochází postupně všechny dvojice záznamů (1. a 2. řádek). Do proměnné *tmp* ukládáme plochu MBR dvou záznamů a v proměnné *largestMbr* si udržujeme dosavadní maximum ploch. Potom porovnáme dosavadní maximum s velikostí plochy dvou záznamů (4. řádek). Pokud je tato plocha větší, tak si aktualizujeme proměnnou *largestMbr* (5. řádek) a proměnné *first* (6. řádek) a *second* (7. řádek), kde *first* je záznam, který bude základem pro první uzel a *second* je záznam, který bude základem pro druhý uzel. Pro zjištění velikosti plochy používáme metodu *ComputeSizeOfMbr* popsanou v algoritmu 4.

Algoritmus 4: ComputeSizeOfMBR

Input: Tuple1. Tuple2
Output: sizeOfMbr

```
1 for i ← 0 to dimension do
2     t1side ← GetUInt(Tuple1, i);
3     t2side ← GetUInt(Tuple2, i);
4     sizeOfMbr ← sizeOfMbr ∗ |(t2side − t1side)|;
5 return sizeOfMbr
```

Tato metoda bere dva parametry a to dva záznamy typu char*. Návratový typ je double. V této metodě se pro každou dimenzi získají dva body MBR (2. a 3. řádek), které nám určují velikost strany MBR v dané dimenzi. Tyto body máme uložené v proměnné *t1side* a *t2side*, jejímž odečtením od sebe získáme velikost strany MBR a následným přinásobením k dosavadní velikosti MBR získáme při projití všech dimenzí velikost celého MBR tvořeného dvěma záznamy *Tuple1* a *Tuple2* (4. řádek). Poté metoda vrací výslednou velikost (5. řádek). Po vytvoření základních dvou záznamů v uzlu a jejich MBR do těchto uzlů začneme přiřazovat další záznamy.

Přiřazovaní zbývajících záznamů máme ukázáno v algoritmu 5. Procházíme záznamy, které jsme dosud nikam nepřiřadili (2. řádek), a počítáme rozdíl ploch mezi zvětšením prvního (3. řádek) a druhého uzlu (4. řádek), které by nastalo při vložení záznamu do uzlu. Rozšíření si uložíme do proměnných *firstDiff*, resp. *secondDiff*. Potom porovnáme dosud největší rozdíl ploch (6. a 11. řádek) udržovaný v proměnné *biggestDiff* s rozdílem ploch *firstDiff* a *secondDiff*. Pokud tento rozdíl bude větší než dosavadní, tak si uložíme nový rozdíl do proměnné *biggestDiff* (7. a 12. řádek) a pomocí proměnné *leafToAdd* si udržujeme záznam s dosavadním největším rozdílem mezi plochami při vložení a v proměnné *mbrToAdd* si udržujeme MBR, do kterého máme záznam přidat. U přiřazení každého dalšího záznamu do uzlu měníme velikost MBR a hodnoty *mbrLo* a *mbrHi* a pokaždé probíhá kontrola, jestli v uzlu už není maximální počet záznamů. Pokud ano, tak zbytek záznamů vložíme vždy do uzlu s menším počtem záznamů. Rozdíly ploch jsem počítal pomocí metody *Enlargement()* ilustrované v algoritmu 6.

Metoda Enlargement bere 3 parametry *mbrLo*, *mbrHi*, *tuple* typu char* a jeden parametr typu double *mbrSize*. První dva parametry definují MBR dosavadního uzlu, další parametr vkládaný záznam a poslední parametr je dosavadní velikost MBR. Metoda má návratový typ

Algoritmus 5: PickNextQuadratic

```
1  for j ← 0 to mItemCount do
2      if GetInt(NodesSelected, j) == 0 then
3          firstDiff ← Enlargement(mbr1Lo, mbr1Hi, GetItem(j), sizeOfFirstMbr);
4          secondDiff ← Enlargement(mbr2Lo, mbr2Hi, GetItem(j), sizeOfSecondMbr);
5          if firstDiff > secondDiff then
6              if biggestDiff < (firstDiff - secondDiff) then
7                  biggestDiff ← firstDiff − secondDiff;
8                  leafToAdd ← j;
9                  mbrToAdd ← 2;
10             else
11                 if biggestDiff < (secondDiff - firstDiff) then
12                     biggestDiff ← secondDiff − firstDiff;
13                     leafToAdd ← j;
14                     mbrToAdd ← 1;
```

Algoritmus 6: Enlargement

```
   Input: mbrLo, mbrHi, tuple, mbrSize
   Output: enlargement
1  for i ← 0 to dimension do
2      lo ← GetUInt(mbrLo, i);
3      hi ← GetUInt(mbrHi, i);
4      tupleSide ← GetUInt(tuple, i, sd);
5      if t1side < t2side then
6          sizeOfMbr ← sizeOfMbr ∗ (hi − tupleSide);
7      else if tupleSide > Hi then
8          sizeOfMbr ← sizeOfMbr ∗ (tupleSide − lo);
9      else
10         sizeOfMbr ← sizeOfMbr ∗ (hi − lo);
11 return sizeOfMbr − mbrSize
```

double a vrací rozšíření MBR. V metodě procházíme všechny dimenze, do proměnných *lo* a *hi* si uložíme *low* (2. řádek) a *high* (3. řádek) MBR v dané dimenzi, do proměnné *tupleSide* si uložíme hodnotu záznamu (4. řádek), pro který simulujeme jeho vložení a počítáme velikost MBR se záznamem podobně jako v metodě *ComputeLargestMBR()* (5. - 10. řádek), poté vrátíme rozdíl velikosti MBR s původní velikostí MBR (11. řádek).

4.2.2 Lineární algoritmus algoritmus

Stejně jako kvadratický algoritmus, lineární algoritmus nejdříve zvolí základní záznamy obou uzlů a potom přerozdělí záznamy. Parametry a návratová hodnota lineárního algoritmu jsou taktéž stejné jako u výše popsaného algoritmu, kde příjmeme MBR obou uzlů, *buffer* se záznamy

v uzlu a *NodesSelected*, do kterého ukládáme rozřazení uzlů. Algoritmus 7 ilustruje vybrání prvních záznamů do každého ze dvou uzlů.

Algoritmus 7: PickSeedsLinear

```
1  for j ← 2 to mItemCount do
2      side ← GetUInt(parent :: GetItem(j, itemBuffer), i, sd);
3      if side > second then
4          second ← side;
5          hightmp ← j;
6      else if side > first then
7          first ← side;
8          lowtmp ← j;
9  if (second - first) > (hiside - lowside) then
10     hiside ← second;
11     lowside ← first;
12     high ← hightmp;
13     low ← lowtmp;
```

V algoritmu 7 hledáme napříč dimenzemi nejnižší *high* hodnotu a nejvyšší *low* hodnotu uzlů, abychom nalezli základní záznamy pro uzly. Do proměnné *second* si ukládáme největší dosavadní hodnotu *high* v dané dimenzi (4. řádek) a v proměnné *first* si ukládáme dosavadní nejmenší nalezenou hodnotu v dané dimenzi (7. řádek). Proměnné *hightmp* a *lowtmp* slouží k uchování informace, který záznam měl nejmenší *low* a největší *high*. Poté porovnáme rozdíl *second* a *first* s rozdílem *hiside* a *loside* (9. řádek), což jsou proměnné, v kterých udržujeme hodnoty *low* a *high* s dosavadním největším rozdílem vzdálenosti mezi nimi. Pokud je rozdíl větší než dosavadní, tak do proměné *hiside* přiřadíme *high* (10. řádek) a do proměnné *lowside* přiřadíme *low* (11. řádek) z této dimenze. V proměnných *high* a *low* udržujeme záznamy s těmito hodnotami a tím se stane tato separace novým hledaným maximem. Po projití všech dimenzí máme nastavené v proměnných *lowside* záznam s nejnižším *low* a v proměnné *hiside* záznam s nejvyšším *high*, které mají největší separaci. Dalším krokem lineárního algoritmu je přerozdělení zbývajících záznamů do uzlů. Přiřazování je ilustrováno v algoritmu 8.

V algoritmu 8 procházíme všechny zbylé záznamy a počítáme opět zvětšení uzlu se záznamem pomocí metody *Enlargement()* popsané v algoritmu 6. Zvětšení prvního uzlu si uložíme do proměnné *firstDiff* (2. řádek) a zvětšení druhého uzlu do proměnné *secondDiff* (3. řádek). Pokud vložení záznamu má za následek větší zvětšení prvního uzlu oproti druhému (4. řádek), tak záznam přiřadíme do druhého uzlu (8. řádek) a upravíme velikost (5. řádek) a rozměry (řádek 6) MBR. V opačném případě přiřadíme záznam do prvního uzlu (13. řádek) a upravíme jeho velikost (10. řádek) a rozměry (11. řádek). Dále musíme opět kontrolovat při každém kroku cyklu, zda v uzlu již není maximální počet záznamů, abychom zbytek záznamů přiřadili do druhého uzlu s menším počtem záznamů.

Algoritmus 8: PickNextLinear

```
1  for i ← 0 to mItemCount do
2      firstDiff ← Enlargement(mbr1Lo, mbr1Hi, GetItem(i), sizeOfFirstMbr);
3      secondDiff ← Enlargement(mbr2Lo, mbr2Hi, GetItem(i), sizeOfSecondMbr);
4      if firstDiff > secondDiff then
5          sizeOfSecondMbr ← sizeOfSecondMbr +
             Enlargement(mbr2Lo, mbr2Hi, GetItem(i, sizeOfSecondMbr);
6          ModifyMbr(mbr2Lo, mbr2Hi, GetItem(i);
7          second + +;
8          SetInt(NodesSelected, i, 2);
9      else
10         sizeOfFirstMbr ← sizeOfFirstMbr +
             Enlargement(mbr1Lo, mbr1Hi, GetItem(i, sizeOfFirstMbr);
11         ModifyMbr(mbr1Lo, mbr1Hi, GetItem(i));
12         first + +;
13         SetInt(NodesSelected, i, 1);
```

4.2.3 Exponenciální algoritmus

Algoritmus 9 hledá všechny kombinace uzlů. Vstupem je číslo p typu int, což je pořadí kombinace, N typu int, což je celkový počet záznamů, z kterých vybíráme, K typu int je počet vybíraných záznamů, *minSize* typu double značí dosavadní nejmenší plochu uzlů, do *combinations* ukládáme další vygenerovanou kombinaci záznamů a do *nodesSelected* ukládáme dosavadní rozdělení záznamů, které ohraničuje nejmenší plochu. Rekurzivní volání metody *Comb()* (5. řádek) nám tedy vygeneruje všechny možné K-prvkové kombinace N záznamů. Pomocí cyklu for (3. řádek) generujeme vzestupné kombinace záznamů, pomocí metody SetInt (4. řádek) určujeme kombinaci.

Algoritmus 9: Comb

```
   Input: p, N, K, minSize, combinations, nodesSelected
1  if p > K then
2      return
3  for i ← GetInt(combinations, p − 1) + 1 to N − K + p do
4      SetInt(combinations, p, i);
5      Comb(p + 1, N, K, minSize, combinations, nodesSelected);
```

Jakmile je $p > K$ (řádek 1), což znamená, že jsme zvolili již kombinaci **K** záznamů. V proměnné *combinations* máme uloženou vygenerovanou jednu kombinaci záznamů. Pro každou kombinaci vytvoříme MBR obou uzlů a poté pomocí metody *ComputeSizeOfMBR()* spočítáme jejich celkovou plochu a porovnáme, jestli je menší, než dosavadní nejmenší plocha MBR.

4.2.4 Greenův rozdělovací algoritmus

U Greenova rozdělovacího algoritmu nejdříve musíme zvolit dva základní záznamy pro původní uzel a pro uzel nový. Zvolení prvních záznamů probíhá stejně jako u algoritmu kvadratického popsaného v algoritmu 3. Dále pro každou dimenzi musíme provést normalizaci těchto záznamů.

Algoritmus 10: ChooseAxis

```
1 for i ← 0 to dimension do
2     separatin ← 0;
3     separation ← (|firstlow − secondhi|)/(GetUInt(mbrHi, i − GetUInt(mbrLo, i));
4     if separation > maxSeparation then
5         maxSeparation ← separation;
6         axis ← i;
```

Projdeme každou dimenzi, v proměnných *firstlow* a *firsthi* udržujeme *low* a *high* prvního záznamu, v proměnných *secondlow* a *secondhi* udržujeme hodnoty *low* a *high* druhého záznamu, které jsme získali z metody *PickSeedsQuadratic()*. Spočteme normalizaci vydělením vzdálenosti dvou uzlů velikostí dimenze (3. řádek). Pokud tato normalizace je větší než dosavadní největší, tak aktualizujeme proměnnou *maxSeparation* (5. řádek) a zvolíme tuto osu jako výchozí pro provedení rozdělení (6. řádek). Setřídíme záznamy v uzlu podle osy uložené v proměnné *axis*. K třídění jsem využil již implementovaný selectsort. Dále přerozdělíme záznamy do uzlů tak, že první půlku záznamu ponecháme v původním uzlu a druhou půlku záznamů vložíme do nového uzlu.

5 Experimenty

Posledním bodem této bakalářské práce je testování naimplementovaných algoritmů a jejich dopad na vlastnosti a propustnost operací v R-stromě. Při testování jsem použil velikost bloku 8192B. Velikost byla použita u všech testovacích kolekcí stejná. Testování na různých kolekcích dat budu provádět pouze pro lineární, kvadratický a Greenův algoritmus a budu je porovnávat s již implementovaných rozdělovacím algoritmem z R*-stromu. Exponenciální algoritmus kvůli jeho exponenciální časové složitosti je nemožné otestovat na těchto kolekcích, tak bude otestován pouze na syntetické kolekci s velikostí bloku 512B.

5.1 Testovací hardware

Naimplementované algoritmy jsem testoval na serveru bayer.cs.vsb.cz s operačním systémem Windows Server 2016 Datacenter a následujícími pamarametry:

1. 512GB RAM
2. Intel Xeon Gold 6136 CPU 3 GHZ

5.2 Testy

Při testování načteme datové kolekce z disku a také načteme následně soubor pro dotazování nad touto kolekcí. U každé testovací kolekce budeme zaznamenávat propustnost operací vkládání a dotazování za sekundu, výšku stromu, průměrnou utilizaci ve vnitřních a v listových uzlech, počet listových a vnitřních uzlů, počet záznamů v listových a vnitřních uzlech a velikost datové struktury. Celé testování probíhá v hlavní paměti.

5.3 Popis testovacích kolekcí

Pro všechny datové kolekce je typ dat unsigned int.

Jméno	Počet záznamů	Dimenze
TIGER[1]	5 889 786	2
CARS[2]	3 360 277	4
IP_TO_ZIP[3]	1 441 818	9
KDDCUP[4]	909 105	42
Syntetická kolekce	1 000 000	4

Tabulka 2: Testovací kolekce dat

[1] http://www.census.gov/geo/www/tiger/
[2] http://www.census.gov/geo/www/tiger/
[3] http://www.infochimps.com/datasets/ip-address-to-zip-code-and-demographics-housing-owner-costs-pt-2
[4] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

U výběru kolekce dat jsem kladl důraz na rozdílnou dimenzi a rozdílný počet záznamů, přičemž všechny kolekce mají dostatečný počet záznamů pro přesnost získaných výsledků. Přehled kolekcí je v tabulce 2.

5.4 Kolekce TIGER

V tabulce 3 můžeme vidět výsledky nad tabulkou TIGER. Výsledky z tabulky 3 ukazují, že

Kolekce dat	Kvadratický	Lineární	Greenův	R*
Vkládání[Záznamů/s]	35 122	218 666	89 043	118 435
Dotazování[Záznamů/s]	79 275	43 242	100 515	94 047
Výška stromu	2	2	2	2
Utilizace vnitřních uzlů[%]	61.2	61.8	64.2	67.6
Utilizace listů[%]	65.4	64.8	66.1	65.4
Vnitřní uzly	53	53	50	48
Listové uzly	13 225	13 354	13 085	13 229
Záznamy ve vnitřních uzlech	13 227	13 406	13 134	13 276
Záznamy v listech	5 889 786	5 889 786	5 889 786	5 889 786
Velikost struktury[MB]	103.73	104.74	102.62	103.73

Tabulka 3: Kolekce TIGER

pokud našim cílem bude dosáhnutí největší propustnosti při vkládání záznamů, nejlepší volbou bude algoritmus lineární. Ovšem při pohledu na rychlost dotazování si vedl lineární algoritmus nejhůře. Kvadratický algoritmus sice oproti lineárnímu algoritmu má dvojnásobnou propustnost dotazování, ale ve srovnání s Greenovým algoritmem zaostává jak co se týče propustnosti vkládání, tak i v propustnosti dotazování. Při porovnání Greenova algoritmu a rozdělení uzlu z R*-stromu tyto algoritmy dosahují podobných výsledků, přičemž Greenův algoritmus dosahuje lepší propustnosti v dotazování, ale horší ve vkládání. Jak můžeme vidět, výška stromu je u všech použitých algoritmů stejná, nejlepší utilizaci listových uzlů dosahuje Greenův algoritmus, zatímco R*-strom rozdělení dosahuje lepší utilizace vnitřních uzlů. Ovšem ve srovnání s ostatními algoritmy zde nejsou příliš velké rozdíly, s utilizací je spojený i nejnižší počet vnitřních a vnějších uzlů a tím i velikost struktury.

5.5 Kolekce CARS

U kolekce CARS vidíme výsledky testů v tabulce 4.

U výsledků z tabulky 4 dosahuje opět největší propustnosti při vkládání algoritmus lineární, nejhůře na tom potom byl algoritmus kvadratický. Greenův algoritmus má podobnou propustnost u dotazování jako algoritmus z R*-stromu, lineární algoritmus za ním zaostává zhruba o polovinu. Kvadratický algoritmus zde dosahuje menší propustnosti než Greenův algoritmus v propustnosti vkládání i dotazování. Rozdělovací algoritmus z R*-stromu má opět lepší propustnost vkládání

Kolekce dat	Kvadratický	Lineární	Greenův	R*
Vkládání[Záznamů/s]	36 684	172 366	96 737	169 514
Dotazování[Záznamů/s]	44 239	37 218	69 835	73 354
Výška stromu	2	2	2	2
Utilizace vnitřních uzlů[%]	62.9	61.5	65.9	64.5
Utilizace listů[%]	64.5	63.9	66.8	65.9
Vnitřní uzly	90	93	83	86
Listové uzly	12 764	12 896	12 333	12 503
Záznamy ve vnitřních uzlech	12 853	12 988	12 415	12 588
Záznamy v listech	3 360 277	3 360 277	3 360 277	3 360 277
Velikost struktury[MB]	100.42	101.48	97.00	98.35

Tabulka 4: Kolekce CARS

oproti Greenovu algoritmu. Utilizaci uzlu má opět nejlepší Greenův algoritmus a s tím spojený i nejnižší počet vnitřních a listových uzlů.

5.6 Kolekce IP_TO_ZIP

V tabulce 5 máme vyslédky dosažené u kolekce IP_TO_ZIP.

Kolekce dat	Kvadratický	Lineární	Greenův	R*
Vkládání[Záznamů/s]	42 994	211 131	105 396	163 642
Dotazování[Záznamů/s]	63 415	29 678	23 608	42 280
Výška stromu	3	3	3	3
Utilizace vnitřních uzlů[%]	47.9	51.2	56.7	59.6
Utilizace listů[%]	57.6	58.7	58.6	61.8
Vnitřní uzly	244	224	202	182
Listové uzly	12 273	12 050	12 055	11 430
Záznamy ve vnitřních uzlech	12 516	12 273	12 256	11 611
Záznamy v listech	1 441 818	1 441 818	1 441 818	1 441 818
Velikost struktury[MB]	97.79	95.89	95.76	90.72

Tabulka 5: Kolekce IP_TO_ZIP

V kolekci IP_TO_ZIP dosahuje opět nejlepších výsledků v propustnosti vkládání algoritmus lineární, za ním je rozdělovací algoritmus z R*-stromu. Greenův algoritmus dosahuje zhruba poloviční výkonnosti, ale překvapivě v propustnosti dotazování je lepší lineární algoritmus oproti Greenovu algoritmu. V dotazování ale uspěl nejlépe algoritmus kvadratický, který má zhruba dvojnásobnou propustnost, dokonce téměř trojnásobnou ve srovnání s Greenovým algoritmem. Co se týká utilizace uzlů tak nejlépe na tom je také algoritmus z R*-stromu.

5.7 Kolekce KDDCUP

Dosažené výsledky u kolekce KDDCUP můžeme vidět v tabulce 6.

Kolekce dat	Kvadratický	Lineární	Greenův	R*
Vkládání[Záznamů/s]	1 0546	1 0942	1 1023	14 263
Dotazování[Záznamů/s]	4 860	5 992	4 238	9 904
Výška stromu	4	4	4	4
Utilizace vnitřních uzlů[%]	62.9	63.8	68.8	60.3
Utilizace listů[%]	64.4	63.8	67.2	67.6
Vnitřní uzly	2 131	2 118	1 855	2 124
Listové uzly	30 043	30 304	28 767	28 608
Záznamy ve vnitřních uzlech	32 173	32 421	30 621	30 731
Záznamy v listech	909 105	909 105	909 105	909 105
Velikost struktury[MB]	251.36	253.30	239.23	240.09

Tabulka 6: Kolekce KDDCUP

Z výsledků zaznamenaných v této tabulce lze vidět téměř stejný výkon v propustnosti u všech algoritmů, přičemž algoritmus lineární zde dosahuje překvapivě lepších výsledků než ostatní algoritmy vyjma rozdělení z R*-stromu. Má při použití této kolekce nejlepší propustnosti vkládání a téměř dvojnásobnou propustnost dotazování oproti ostatním algoritmům. Snížená propustnost operací oproti ostatním kolekcím může být také z důvodu zvýšení výšky R-stromu a také počet uzlů je několikanásobně větší než u předchozích kolekcí. Utilizaci uzlů dosahuje nejlepší Greenův algoritmus a tím také dosahuje nejlepší velikosti struktury.

5.8 Testování exponenciálního algoritmu

Testování exponenciálního algoritmu jsem prováděl na náhodně vygenerované kolekci pomocí cTuplesGenerator, kde jsem volil pouze velikost bloku 512B a počet záznamů jsem volil 1 000 000 kvůli časové složitosti tohoto algoritmu.

Jak vidíme v tabulce 7, tak exponenciální algoritmus dosahuje naprosto hrozných výsledků ve všech faktorech. Rychlost vkládání je pouze 10 záznamů za sekundu, což je nereálné pro praktické použití. Výška stromu stoupá rychle kvůli plnění uzlů, které mají malou kapacitu, kapacitu je nutné volit nízkou kvůli exponenciální složitosti. Utilizace uzlů je sice dobrá, nicméně to nemění nic na nepoužitelnosti tohoto algoritmu. I ostatní algoritmy si nevedly příliš dobře co se týče propustnosti při malé kapacitě uzlů.

Kolekce dat	Exponenciální algoritmus	Lineární	KVadratický	Greenův
Vkládání[Záznamů/s]	10	1626	825	1560
Dotazování[Záznamů/s]	3 864	804	2 848	888
Výška stromu	5	6	6	5
Utilizace vnitřních uzlů[%]	70.7	62.1	60.7	69.5
Utilizace listů[%]	67.8	64.0	65.3	67.5
Vnitřní uzly	7 504	9 203	9 271	7 681
Listové uzly	61 426	65 075	63 845	61 760
Záznamy ve vnitřních uzlech	68 929	74 277	73 115	69 440
Záznamy v listech	1 000 000	1 000 000	1 000 000	1 000 000
Velikost struktury[MB]	33.66	36.27	35.70	33.91

Tabulka 7: Kolekce dat vygenerovaná pomocí náhodného generátoru

6 Závěr

Cílem této práce bylo nastudovat datovou strukturu R-strom, rozdělovací algoritmy pro rozdělení uzlu, začlenit vybrané algoritmy do frameworku RadegastDB a následně otestovat a porovnat rozdíly ve výkonu datové struktury při použití implementovaných algoritmů. Tyto cíle byly dosaženy a byly implementovány algoritmy navrhnuté při uvedení R-stromu, tedy kvadratický, lineární a exponenciální a dále algoritmus Greenův. Tyto algoritmy jsem následně testoval a porovnával jejich dopad na R-strom. Lineární algoritmus se ukázal jako nejlepší varianta pokud nám jde především o propustnost vkládání, ovšem co se týče propustnosti u dotazování, tak si vedl nejhůře. Utilizace uzlů u lineárního algoritmu byla také nejmenší, a tím větší je také velikost struktury. Kvadratický algoritmus nedosahoval přilíš dobrých výsledků u propustnosti vkládání, ovšem co se týče propustnosti u dotazování, tak si vedl lépe, rozdělení záznamů do uzlů je lepší a s tím spojená utilizace uzlů. Velikost struktury je menší než při použití lineárního algoritmu. Exponenciální algoritmus ve výsledku dopadl nejhůře ze všech. Při propustnosti vkládání dosažené u tohoto algoritmu je nepoužitelný v praxi. Dotazování již probíhalo o něco lépe, ale kvůli navyšujícímu se počtu vnitřních uzlů a výšce stromu jsem dostával i u dotazování špatné výsledky. Větší počet vnitřních uzlů souvisí se zvyšující se velikostí struktury. Greenův algoritmus dosahoval v porovnání z ostatními algoritmy dobrých výsledků jak při dotazování, tak i při vkládání záznamů. Utilizace v tomto případě byla nejlepší a tím i datová struktura byla nejmenší. Nejlepších výsledků dosahoval původně implementovaný rozdělovací algoritmus z R*-stromu, přičemž u většiny kolekcí dosahoval pouze o něco horších výsledků Greenův algoritmus. Tato bakalářská práce poskytla přehled o výkonu R-stromu při použití prvních algoritmů uvedených s R-stromem. V dnešní době se přichází s již sofistikovanějšími algoritmy, u kterých se dosahuje lepších výsledků.

Literatura

[1] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 47–57. ACM, 1984.

[2] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Proceedings of the 13th International Conference on Very Large Data Bases, Pages 507-518, 1987

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R-tree: An Efficient and Robust Access Method for Points and Rectangles. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 322–331. ACM Press, 1990

[4] Ing. Peter Chovanec, Ph.D. REDUCTION OF DISK ACCESSES IN MULTIDIMENSIONAL DATA STRUCTURES, Dizertační práce VŠB – Technical University of Ostrava Faculty of Electrical Engineering and Computer Science Department of Computer Science, Ostrava 2015.

[5] BHATTACHARYA, Arnab. Fundamentals of database indexing and searching. Boca Raton: Taylor Francis, [2015]. ISBN 9781466582545.

[6] MANOLOPOULOS, Yannis. R-trees: theory and applications. London: Springer, c2006. Advanced information and knowledge processing. ISBN 9781852339777.

[7] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), pages 500–509. Morgan Kaufmann Publishers Inc., 1994.

[8] STR: A Simple and Efficient Algorithm for R-Tree Packing [online]. [cit. 2019-04-23]. Dostupné z: http://wayback.archive-it.org/1792/20100513142528/http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19970016975_1997

[9] Corner-based splitting: An improved node splitting algorithm for R-tree [online]. [cit. 2019-04-23]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.925.4498rep=rep1type=pdf

[10] LINEAR R-TREE REVISITED [online]. [cit. 2019-04-23]. Dostupné z: https://pdfs.semanticscholar.org/d1e6/ed86efab4fe2864f70c99806defff4d2c6a5.pdf